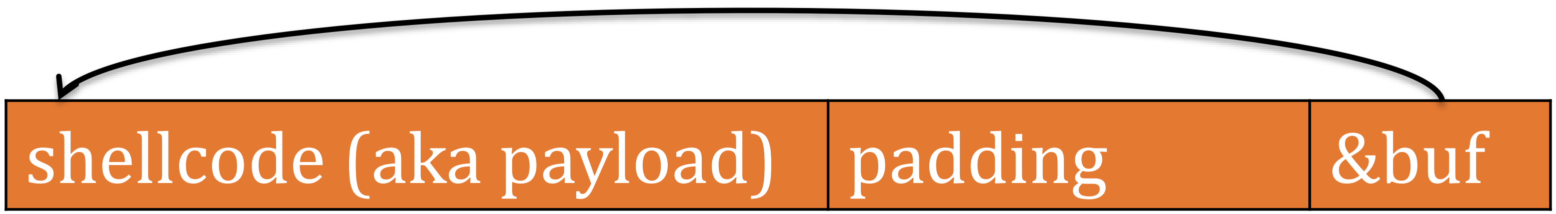
Control Flow Hijack Defenses

Canaries, DEP, and ASLR

David Brumley
Carnegie Mellon University

Control Flow Hijack: Always control + computation



computation + ***control***

- code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- ...

Same principle, different mechanism

Control Flow Hijacks

… happen when an attacker gains control of ***the instruction pointer****.*

Two common hijack methods:

- buffer overflows
- format string attacks

Control Flow Hijack Defenses

Bugs are the root cause of hijacks!

- Find bugs with analysis tools
- Prove program correctness

Mitigation Techniques:

- Canaries
- Data Execution Prevention/No eXecute
- Address Space Layout Randomization

Proposed Defense Scorecard

Aspect	Defense
Performance	• Smaller impact is better
Deployment	• Can everyone easily use it?
Compatibility	• Doesn't break libraries
Safety Guarantee	• Completely secure to easy to bypass

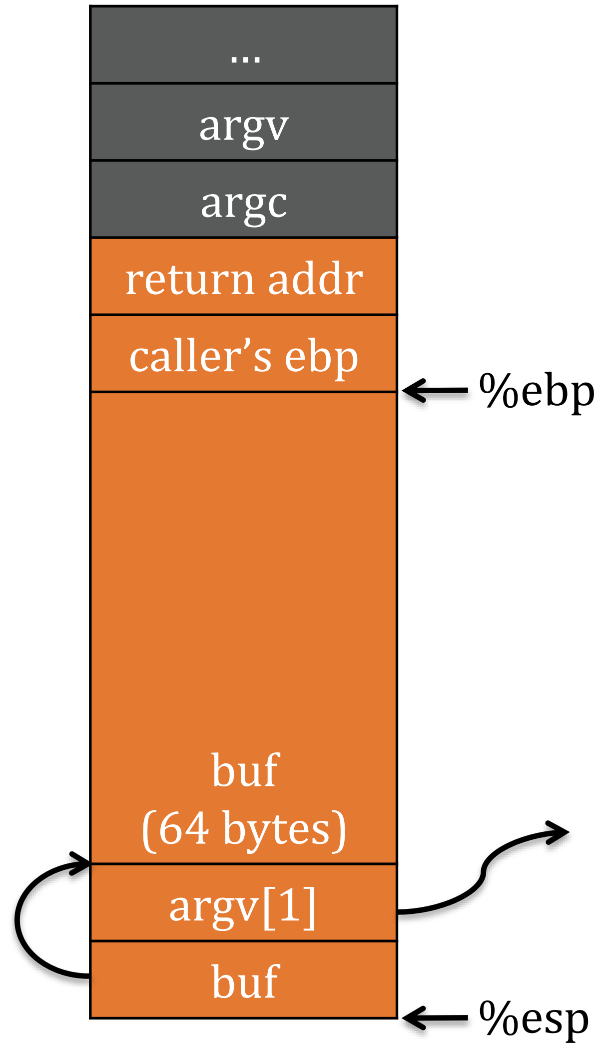
* http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx

Canary / Stack Cookies

http://en.wikipedia.org/wiki/File:Domestic_Canary_-_Serinus_canaria.jpg

“A”x68 . “\xEF\xBE\xAD\xDE”

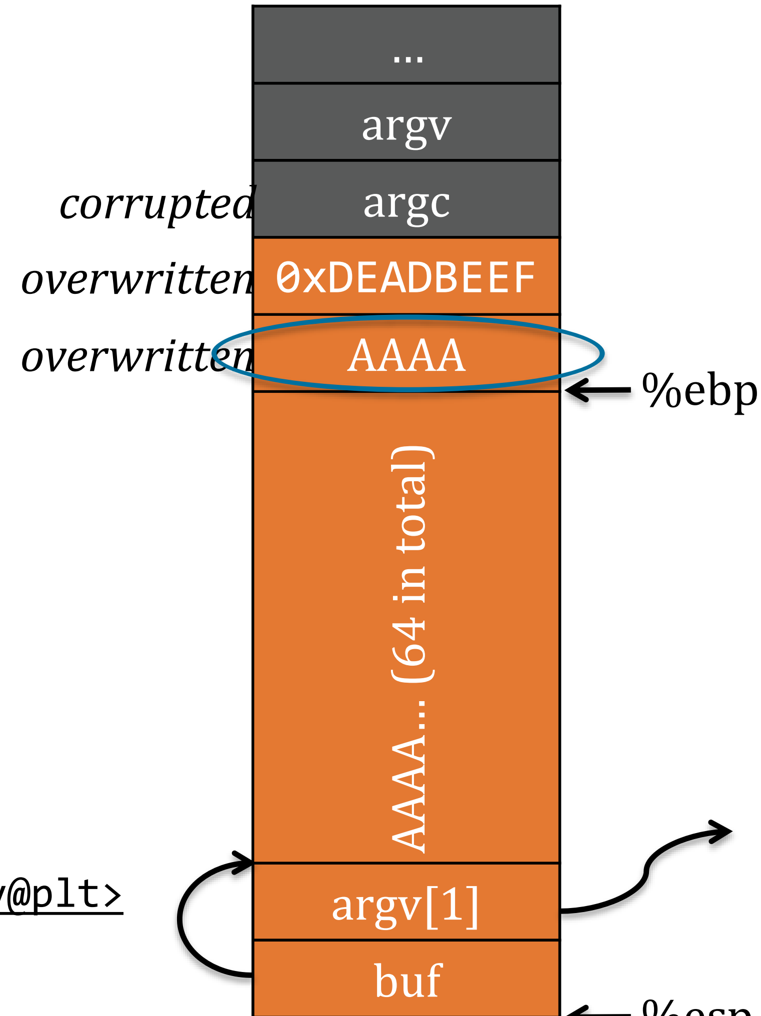
```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

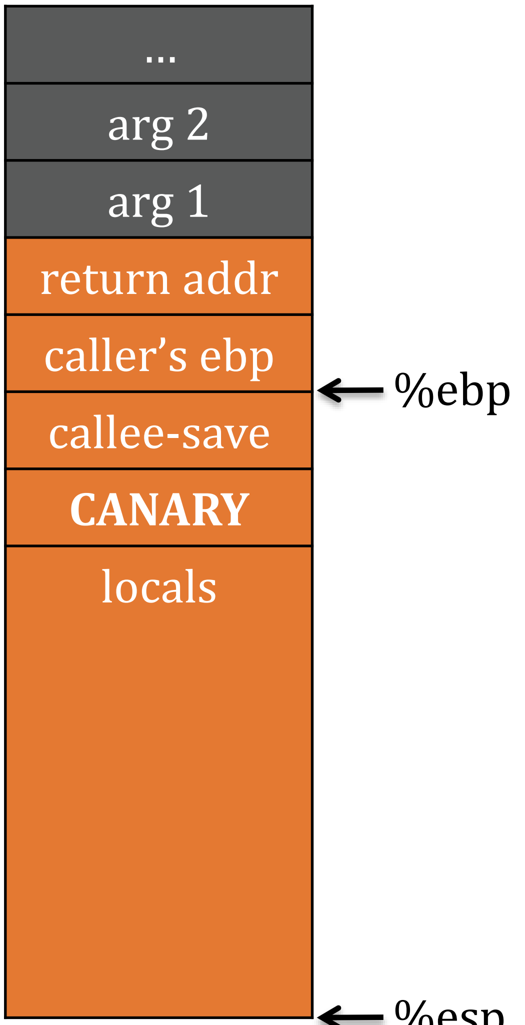


StackGuard [Cowen etal. 1998]

Idea:

- prologue introduces a ***canary word*** between return addr and locals
- epilogue checks canary before function returns

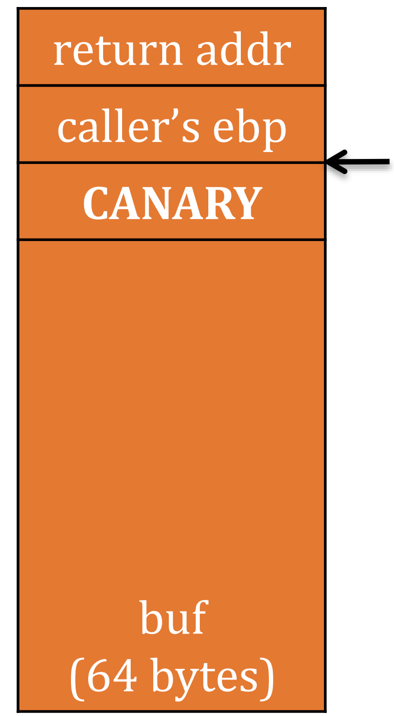
Wrong Canary => Overflow



gcc Stack-Smashing Protector (ProPolice)

```
Dump of assembler code for function main:
   0x08048440 <+0>:  push   %ebp
   0x08048441 <+1>:  mov    %esp,%ebp
   0x08048443 <+3>:  sub    $76,%esp
   0x08048446 <+6>:  mov    %gs:20,%eax
   0x0804844c <+12>: mov    %eax,-4(%ebp)
   0x0804844f <+15>: xor    %eax,%eax
   0x08048451 <+17>: mov    12(%ebp),%eax
   0x08048454 <+20>: mov    4(%eax),%eax
   0x08048457 <+23>: mov    %eax,4(%esp)
   0x0804845b <+27>: lea    -68(%ebp),%eax
   0x0804845e <+30>: mov    %eax,(%esp)
   0x08048461 <+33>: call   0x8048350 <strcpy@plt>
   0x08048466 <+38>: mov    -4(%ebp),%edx
   0x08048469 <+41>: xor    %gs:20,%edx
   0x08048470 <+48>: je     0x8048477 <main+55>
   0x08048472 <+50>: call   0x8048340 <__stack_chk_fail@plt>
   0x08048477 <+55>: leave
   0x08048478 <+56>: ret
```

Compiled with v4.6.1:
`gcc -fstack-protector -O1 …`



Canary should be **HARD** to Forge

- Terminator Canary
  - 4 bytes: 0,CR,LF,-1 (low->high)
  - terminate `strcpy()`, `gets()`, …

- Random Canary
  - 4 random bytes chosen at load time
  - stored in a guarded page
  - need good randomness

Canary Scorecard

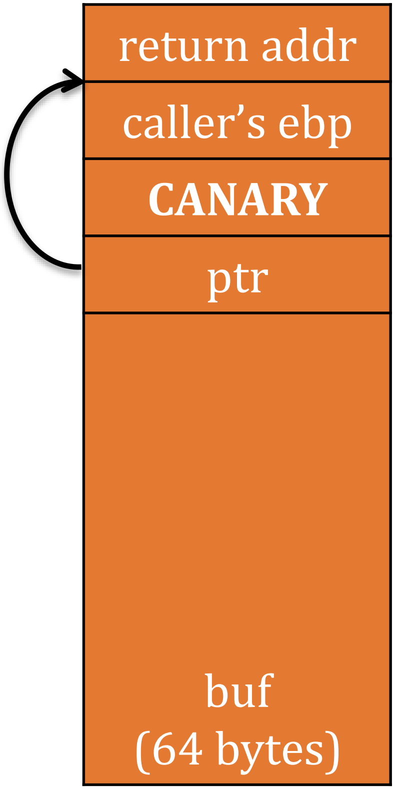
Aspect	Canary
Performance	• several instructions per function • time: a few percent on average • size: can optimize away in safe functions (but see MS08-067 *)
Deployment	• recompile suffices; no code change
Compatibility	• perfect—invisible to outside
Safety Guarantee	• *not really…*

* http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx

Bypass: Data Pointer Subterfuge

Overwrite a data pointer *first*…

```
int *ptr;
char buf[64];
memcpy(buf, user1);
*ptr = user2;
```



Canary Weakness

Check does ***not*** happen until epilogue…

- func ptr subterfuge } PointGuard
- C++ vtable hijack
- exception handler hijack } SafeSEH SEHOP
- …

ProPolice puts arrays above others *when possible*

`struct` is fixed;
& what about heap?

Code Examples:
http://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx

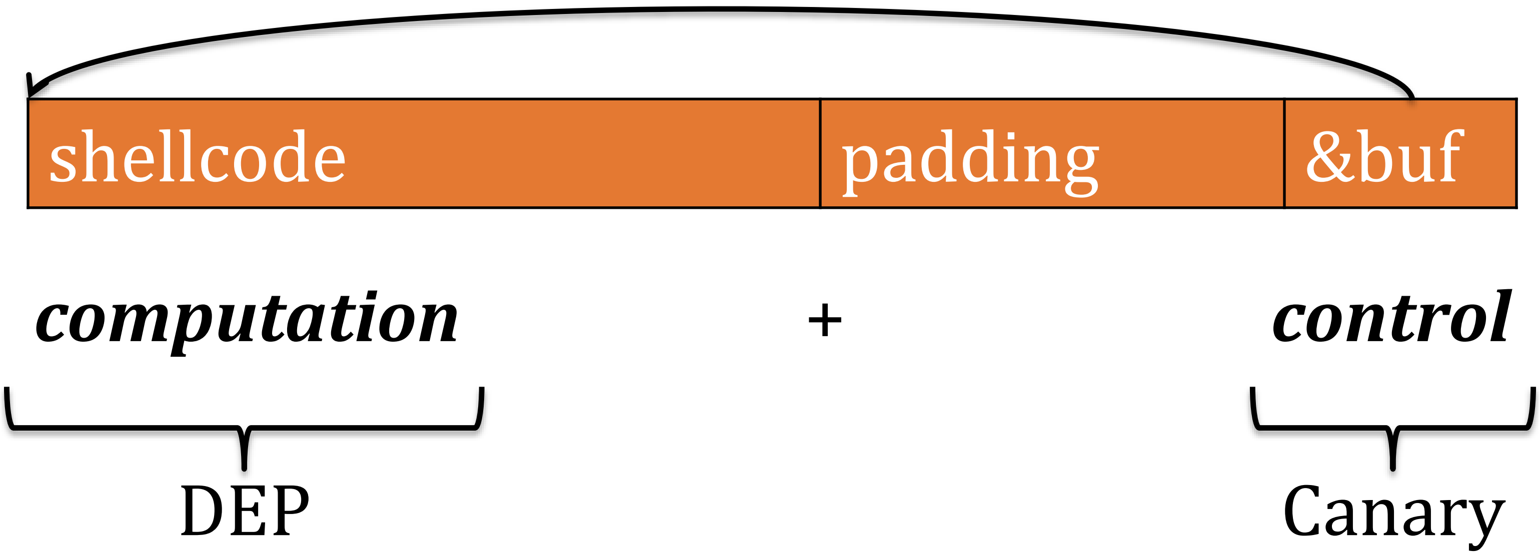
VS 2003: /GS

What is "Canary"?

Wikipedia: "the historic practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system."

Data Execution Prevention (DEP) / No eXecute (NX)

How to defeat exploits?

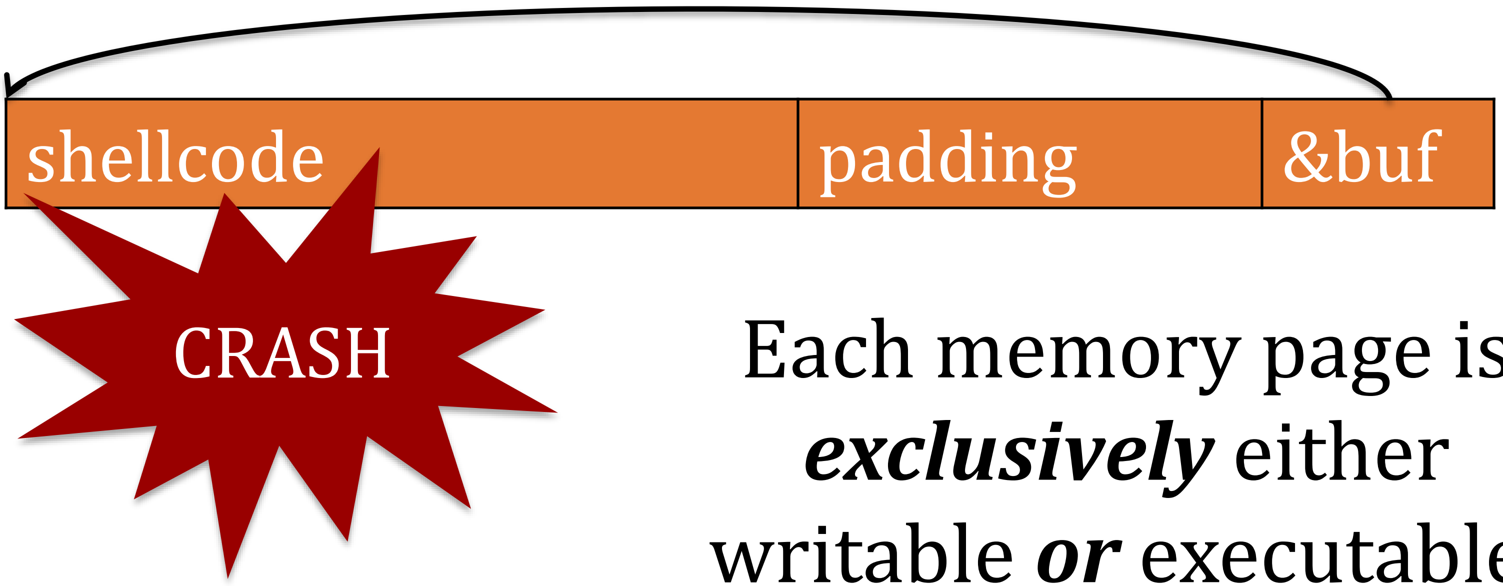


Data Execution Prevention

Mark stack as non-executable using NX bit

(still a Denial-of-Service attack!)

W ^ X



Each memory page is ***exclusively*** either writable ***or*** executable.

(still a Denial-of-Service attack!)

DEP Scorecard

Aspect	Data Execution Prevention
Performance	• with hardware support: no impact • otherwise: reported to be <1% in PaX
Deployment	• kernel support (common on all platforms) • modules opt-in (less frequent in Windows)

Compatibility	• can break legitimate programs
	  - Just-In-Time compilers
	  - unpackers

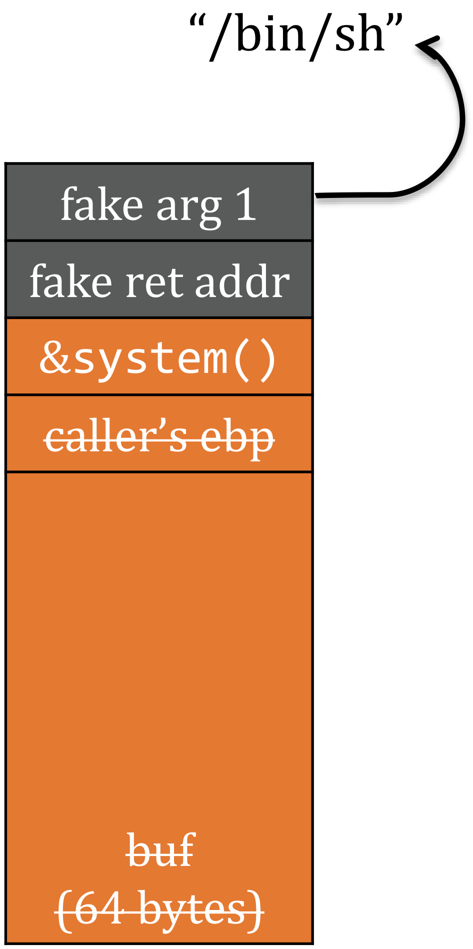
Safety Guarantee	• code injected to NX pages never execute • *but code injection may not be necessary…*

Return-to-libc Attack

Overwrite return address by address of a libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!

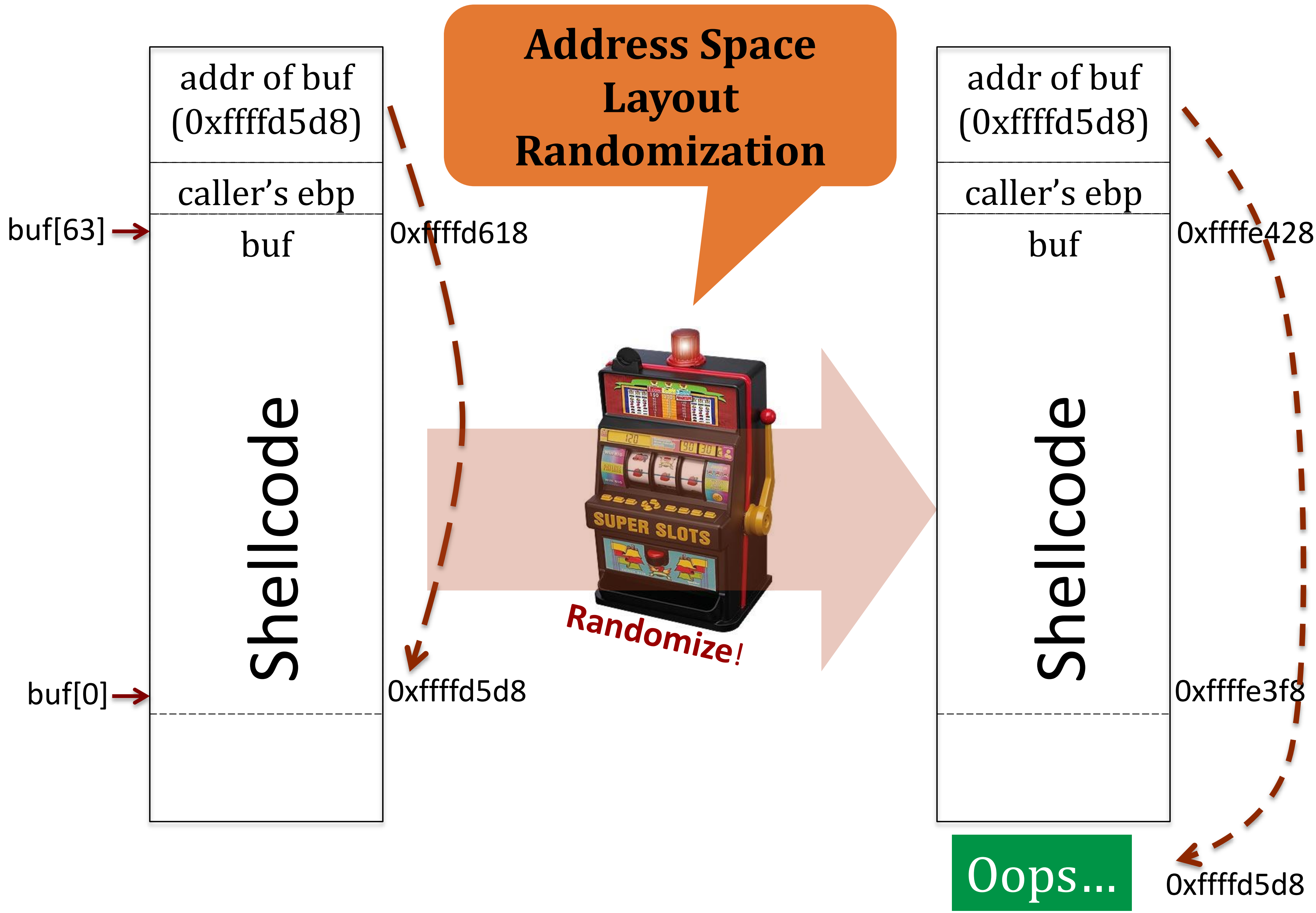


More to come later

Address Space Layout Randomization (ASLR)

Assigned Reading:

ASLR Smack and Laugh Reference
by Tilo Muller


Address Space Layout Randomization
addr of buf (0xffffd5d8)
caller's ebp
buf
Shellcode
buf[63]
buf[0]
0xffffd618
0xffffd5d8
Randomize!
SUPER SLOTS
addr of buf (0xffffd5d8)
caller's ebp
buf
Shellcode
0xffffe428
0xffffe3f8
Oops…
0xffffd5d8

ASLR

Traditional exploits need precise addresses

- *stack-based overflows:* location of shell code
- *return-to-libc:* library addresses

- **Problem:** program's memory layout is fixed
  - stack, heap, libraries etc.

- **Solution:** randomize addresses of each region!

Running cat Twice

- Run 1

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000 rw-p 082ac000 00:00 0          [heap]
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf966000-bf97b000 rw-p bffeb000 00:00 0          [stack]
```

- Run 2

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
086e8000-08709000 rw-p 086e8000 00:00 0          [heap]
b7d9a000-b7eef000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7eef000-b7ef0000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7ef0000-b7ef2000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf902000-bf917000 rw-p bffeb000 00:00 0          [stack]
```

Memory

Base address a

Program
- Code
- Uninitialized data
- Initialized data

Base address b

Mapped
- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Base address c

Stack
- Main stack

ASLR Randomization

a **+ 16 bit rand r_1** b **+ 16 bit rand r_2** c **+ 24 bit rand r_3**

Program

- Code
- Uninitialized data
- Initialized data

Mapped

- Heap
- Dynamic libraries
- Thread stacks
- Shared Memory

Stack

- Main stack

* ≈ 16 bit random number of 32-bit system. More on 64-bit systems.

ASLR Scorecard

Aspect	Address Space Layout Randomization
Performance	• excellent—randomize once at load time
Deployment	• turn on kernel support (Windows: opt-in per module, but system override exists) • no recompilation necessary
Compatibility	• transparent to safe apps (position independent)
Safety Guarantee	• not good on x32, much better on x64 • *code injection may not be necessary…*

Ubuntu - ASLR

- ASLR is **ON** by default [Ubuntu-Security]
  – cat /proc/sys/kernel/randomize_va_space
    - Prior to Ubuntu 8.10: **1** *(stack/mmap ASLR)*
    - In later releases: **2** *(stack/mmap/brk ASLR)*

– stack/mmap ASLR: since kernel 2.6.15 (Ubuntu 6.06)

– brk ASLR: since kernel 2.6.26 (Ubuntu 8.10)

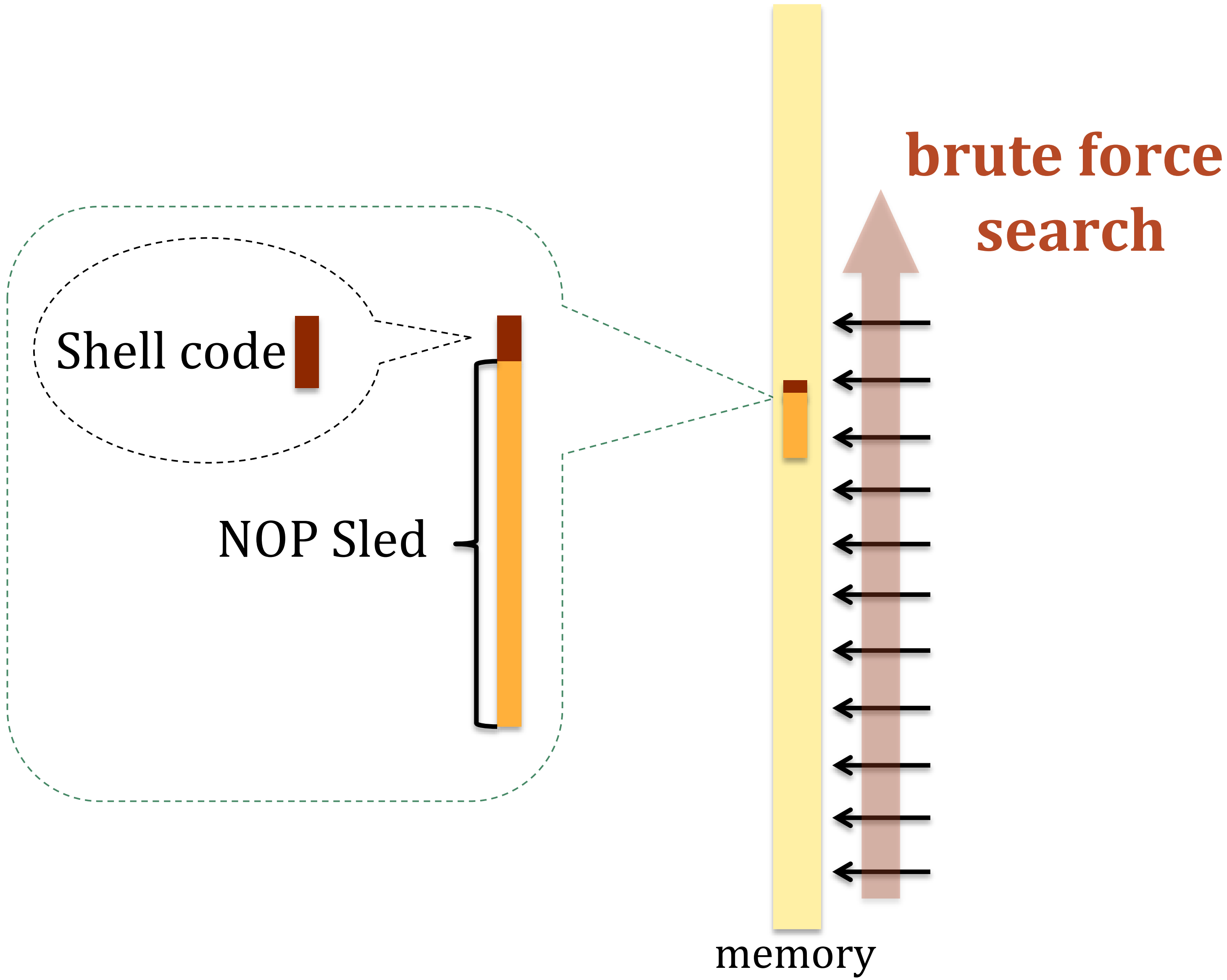
– exec ASLR: since kernel 2.6.25
  - Position Independent Executable (PIE) with “-fPIE –pie”

How to attack with ASLR?

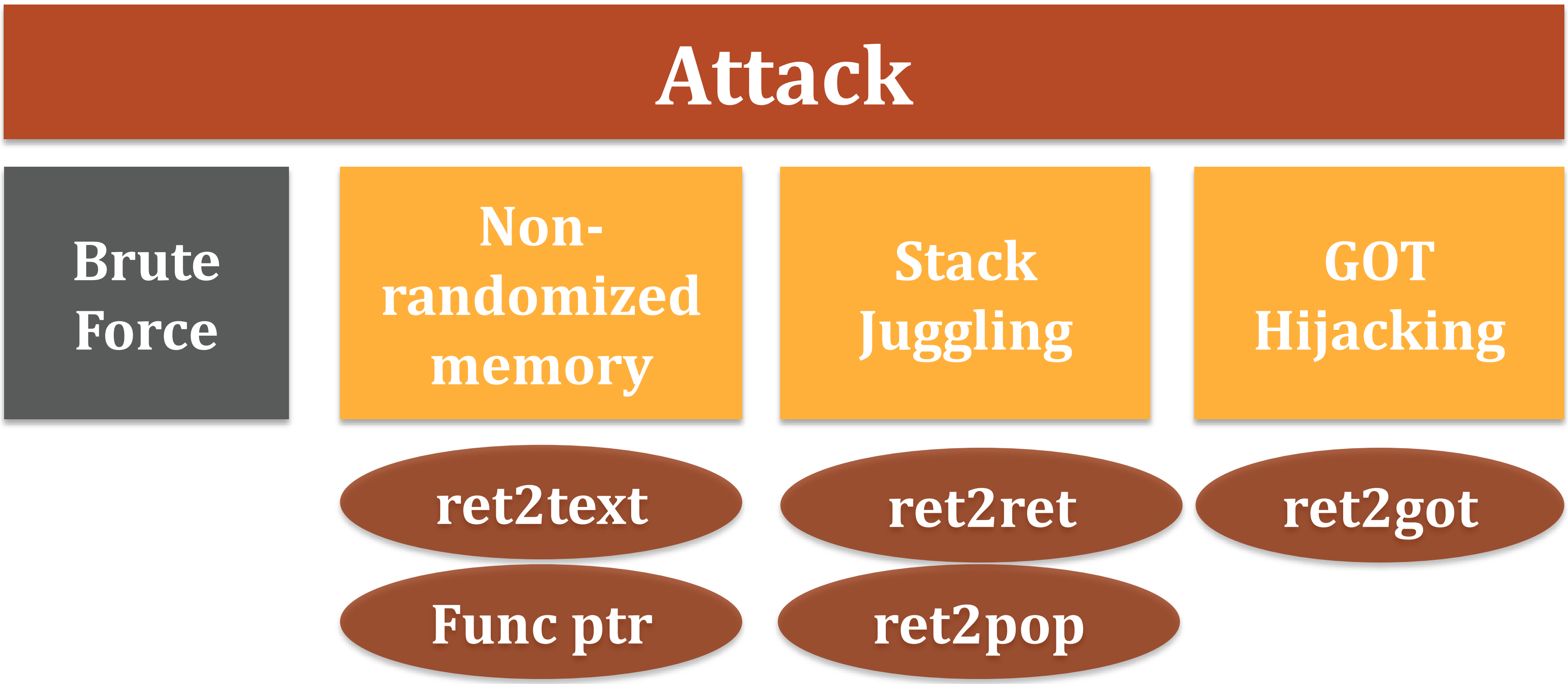
Attack

- **Brute Force**
- **Non-randomized memory**
  - ret2text
  - Func ptr
- **Stack Juggling**
  - ret2ret
  - ret2pop
- **GOT Hijacking**
  - ret2got

Brute Force



How to attack with ASLR?



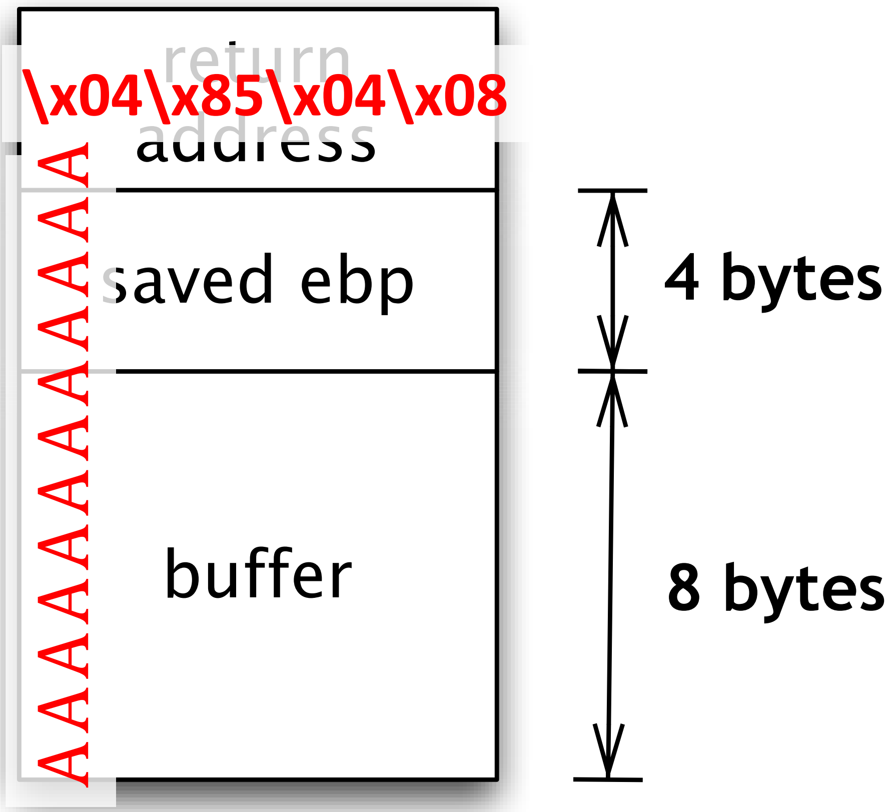
ret2text

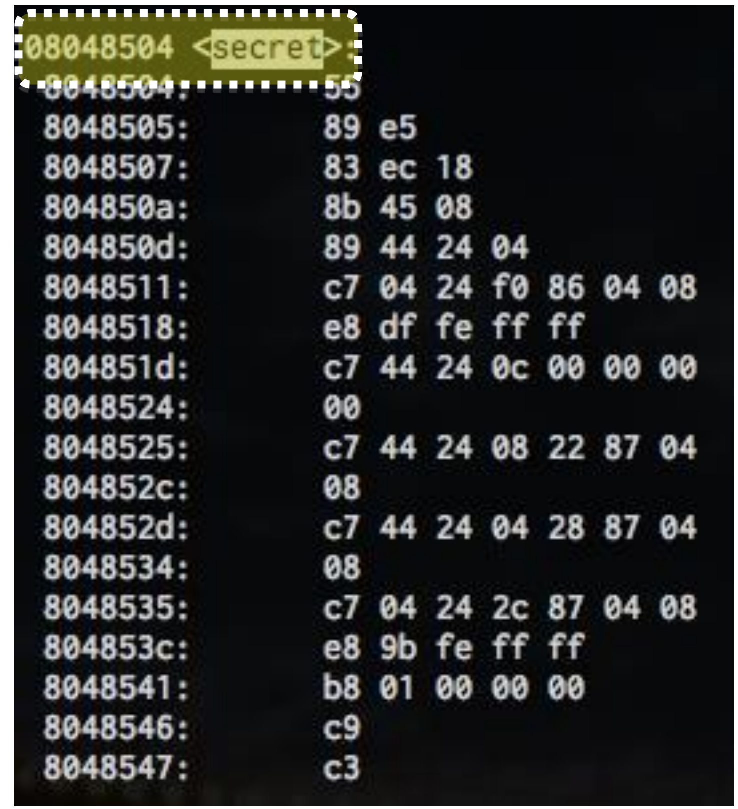
- `text` section has executable program code
  – but not typically randomized by ASLR except PIE

- can hijack control flow to unintended (but existing) program function
  – Figure 7 in reading

ret2text

.text not randomized





Same as running "winner" in vuln2 from class exercise

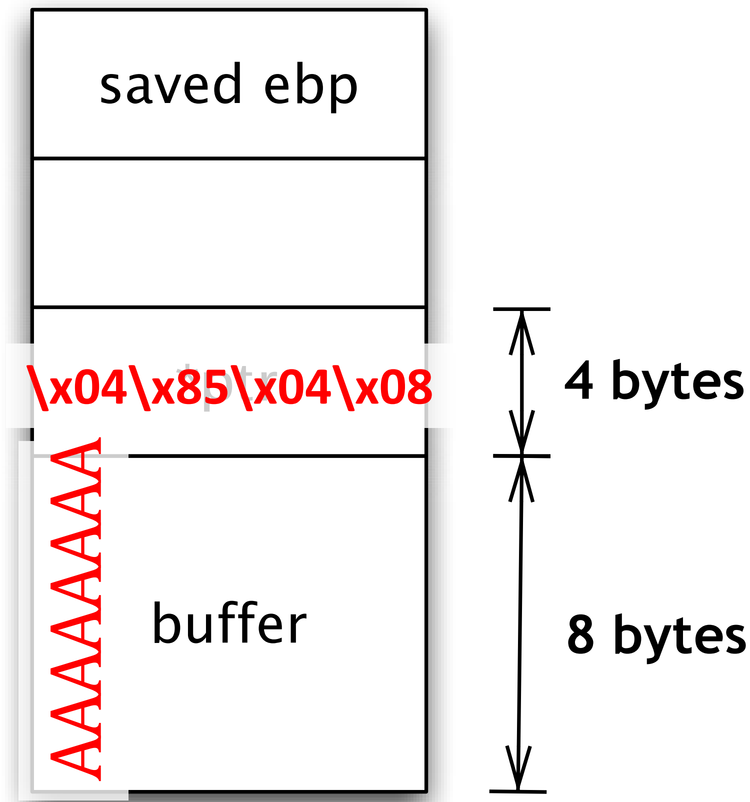
Function Pointer Subterfuge

Overwrite a function pointer to point to:

- program function (similar to ret2text)
- another lib function in Procedure Linkage Table

```
/*please call me!*/
int secret(char *input) { … }

int chk_pwd(char *intput) { … }

int main(int argc, char *argv[]) {
    int (*ptr)(char *input);
    char buf[8];

    ptr = &chk_pwd;
    strncpy(buf, argv[1], 12);
    printf("[] Hello %s!\n", buf);

    (*ptr)(argv[2]);
}
```

Function Pointers




```
08048504 <secret>:
 8048504:       55
 8048505:       89 e5
 8048507:       83 ec 18
 804850a:       8b 45 08
 804850d:       89 44 24 04
 8048511:       c7 04 24 30 87 04 08
 8048518:       e8 df fe ff ff
 804851d:       c7 44 24 0c 00 00 00
```

```
ptr = &chk_pwd;
strncpy(buf, argv[1], 12);
printf("[] Hello %s!\n", buf);

(*ptr)(argv[2]);
```

How to attack with ASLR?

Attack

- **Brute Force**
- **Non-randomized memory**
  - ret2text
  - Func ptr
- **Stack Juggling**
  - ret2ret
  - ret2pop
- **GOT Hijacking**
  - ret2got

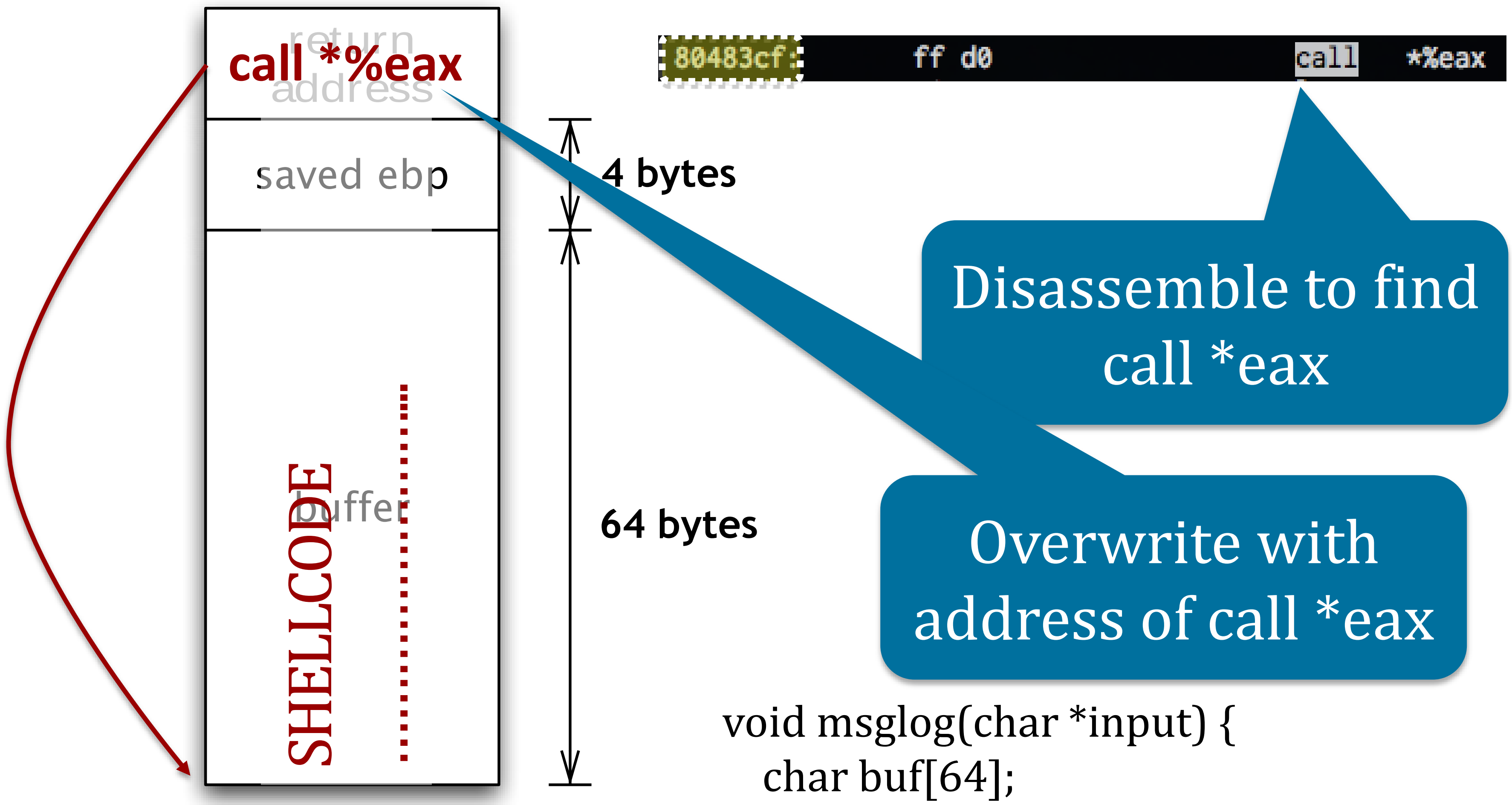
ret2eax

```
void msglog(char *input) {
    char buf[64];
    strcpy(buf, input);
}

int main(int argc, char *argv[]) {
    if(argc != 2) {
        printf("exploitme <msg>\n");
        return -1;
    }

    msglog(argv[1]);

    return 0;
}
```

returns pointer to buf in eax

A subsequent call *eax would redirect control to buf

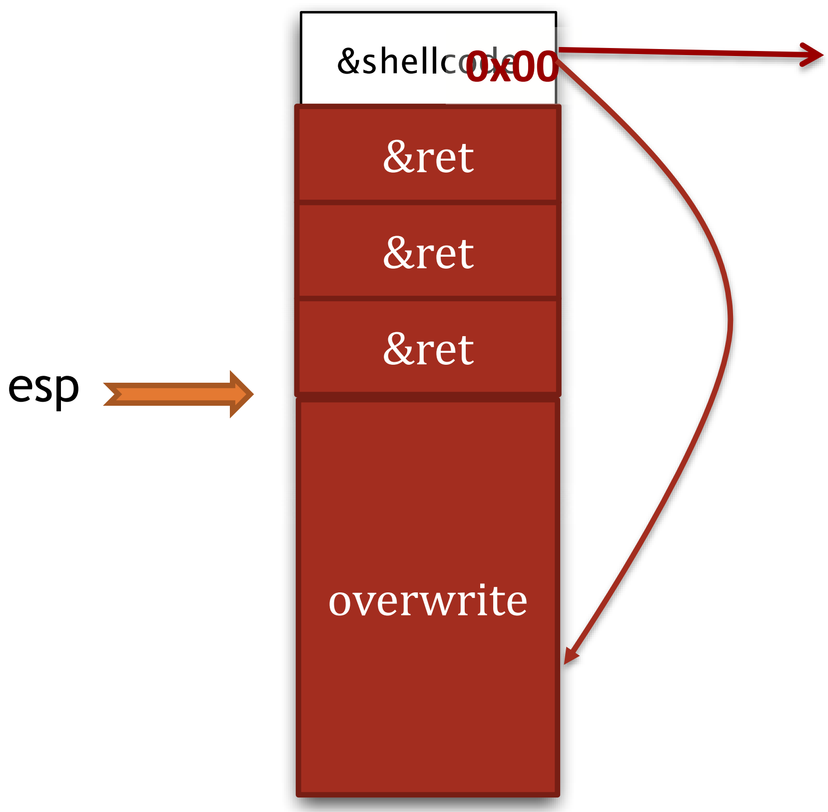
ret2eax




```
void msglog(char *input) {
  char buf[64];
  strcpy(buf, input);
}
```

ret2ret

- If there is a valuable (***potential shellcode***) **pointer** on a stack, you might consider this technique.



shellcode (usually resides in buf, but how to point there?)

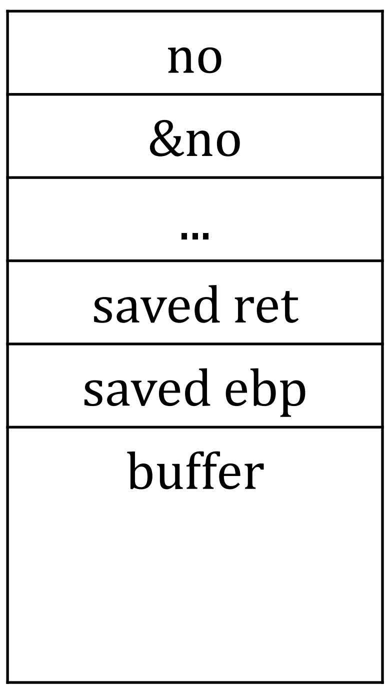
ret = pop eip; jmp eip;

“stack juggling”

ret2ret (stack juggling)

You might consider this technique when

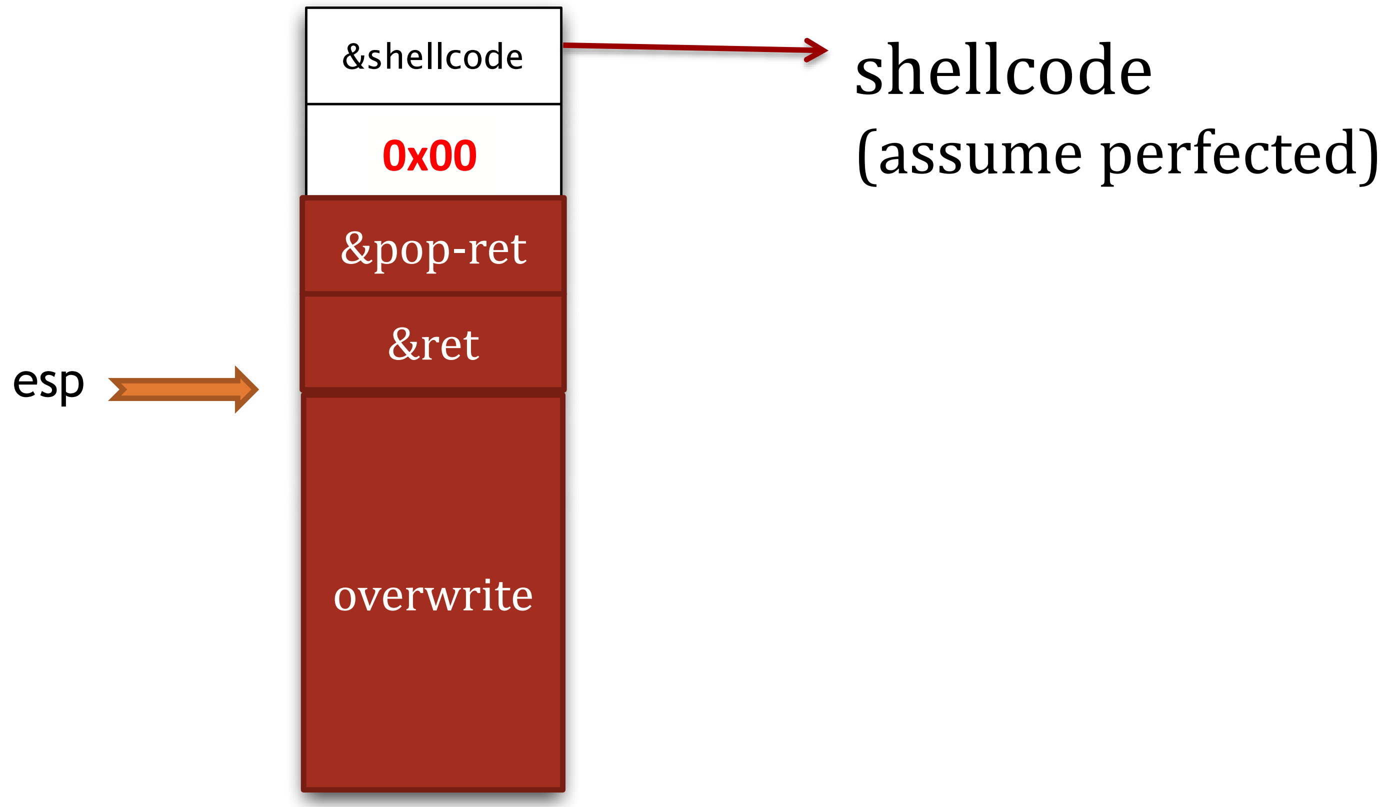
- Text section isn't randomized (uses addr of ret instr)
- Can overwrite pointer `ptr` that points to stack
- `ptr` is higher on the stack than vuln `buffer`



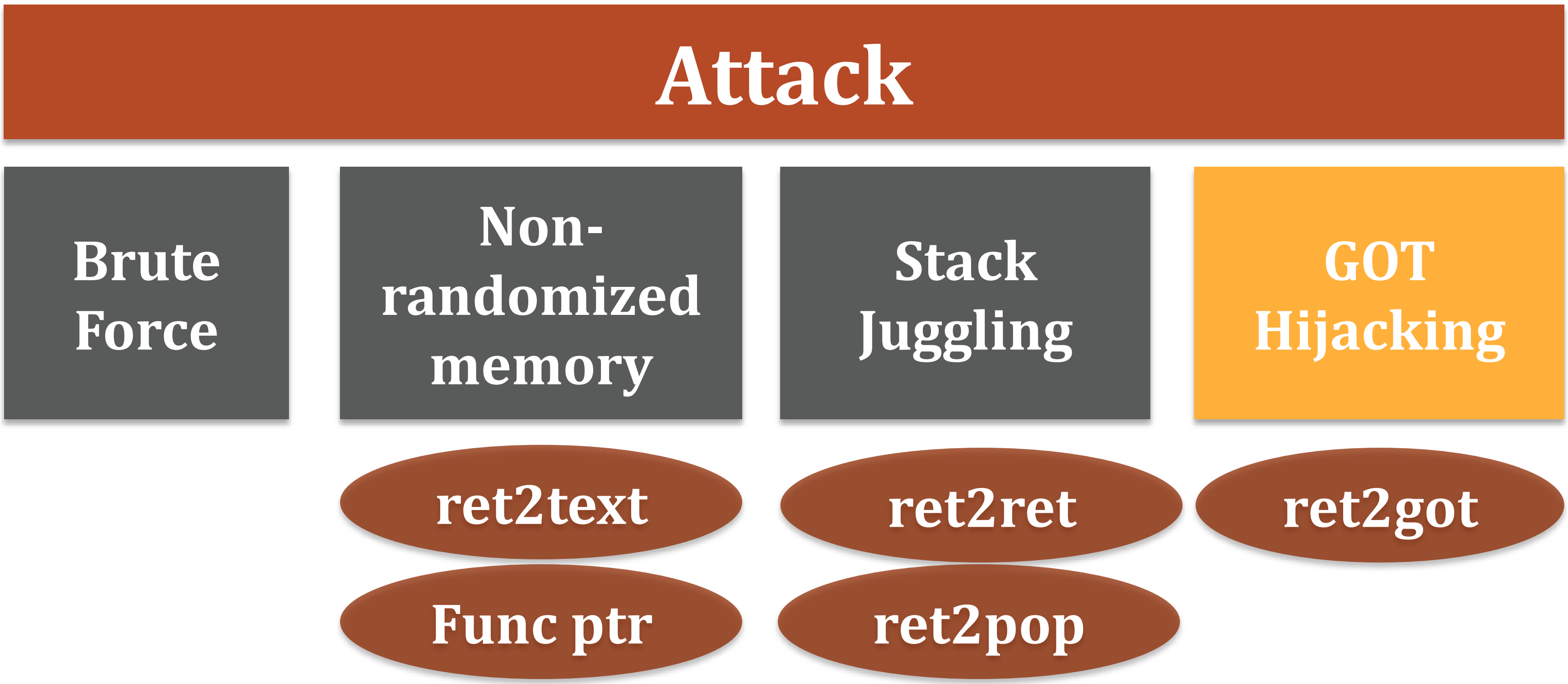

```
void f(char *str) {
    char buffer[256];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    int no = 1;
    int *ptr = &no;
    f(argv[1]);
}
```

ret2pop

- If there is a valuable (***potential shellcode***) **pointer** on a stack, you might consider this technique.



How to attack with ASLR?



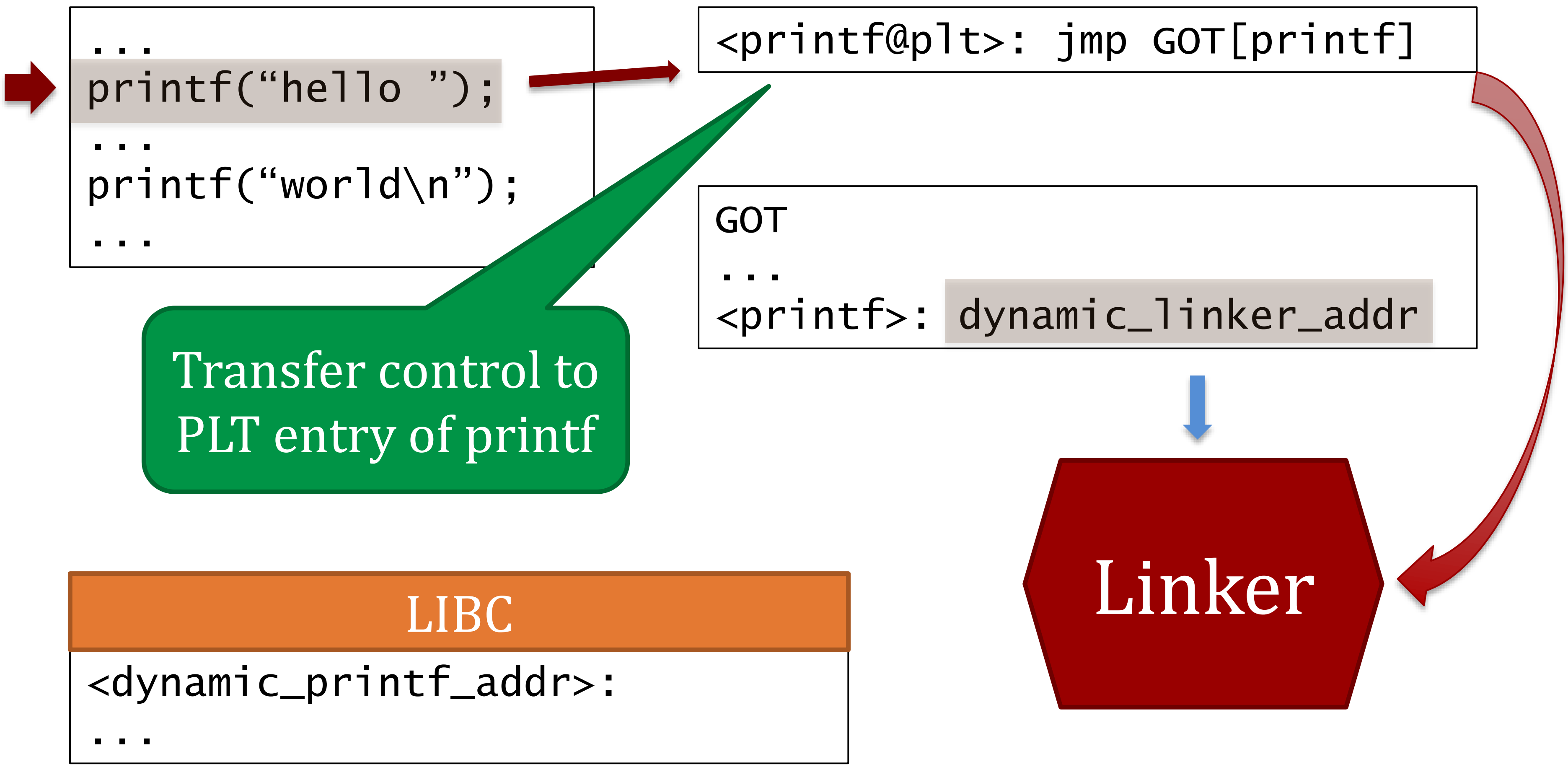
Other Non-randomized Sections

- Dynamically linked libraries are loaded at runtime. This is called ***lazy binding***.

- Two important data structures
  - Global Offset Table
  - Procedure Linkage Table

} commonly positioned statically at compile-time

Dynamic Linking



Dynamic Linking

```
...
printf("hello ");
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_printf_addr
```

Linker fills in the actual addresses of library functions

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

Dynamic Linking

```
...
printf("hello ");
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_printf_addr
```

Subsequent calls to printf do not require the linker

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

Exploiting the linking process

- GOT entries are really function pointers positioned at known addresses

- **Idea:** use other vulnerabilities to take control (e.g., format string)

GOT Hijacking

```
...
printf(usr_input);
...
printf(“world\n”);
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_linker_addr
```

Use the format string to overwrite a GOT entry

```
LIBC
<dynamic_printf_addr>:
...
```

Linker

GOT Hijacking

```
...
printf(usr_input);
...
printf("world\n");
...
```

Use the format string to overwrite a GOT entry

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: any_attacker_addr
```

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

GOT Hijacking

```
...
printf(usr_input);
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: any_attacker_addr
```

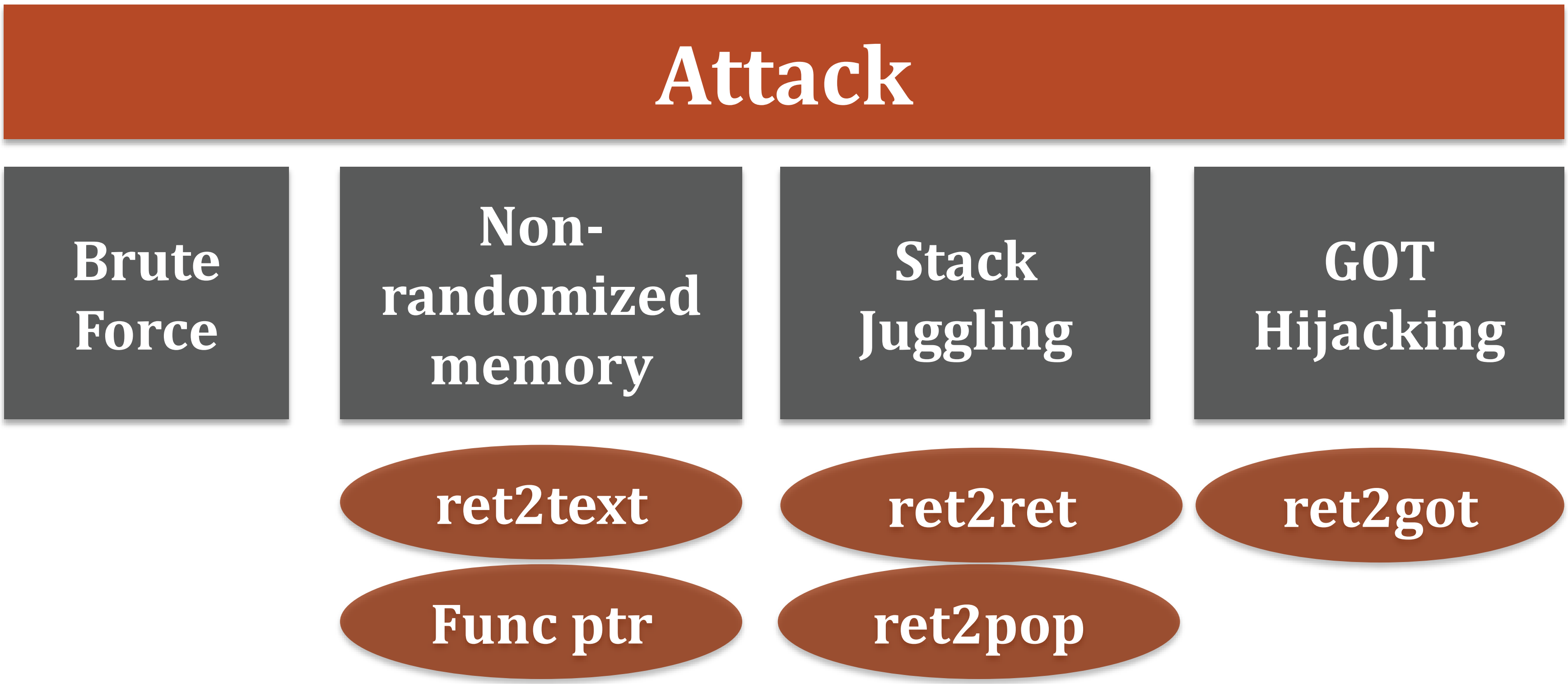
The next invocation transfers control wherever the attacker wants (e.g., system, pop-ret, etc)

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

How to attack with ASLR?



Many other techniques

- ret2bss, ret2data, ret2heap, ret2eax
- string pointer
- ret2dtors
  - overwriting dtors section

The Security of ASLR

Optional Reading:

On the Effectiveness of Address-Space Randomization
by Shacham et al, ACM CCS 2004

```
$ /bin/cat /proc/self/maps
08048000-0804f000 r-xp 00000000 08:01 2514948    /bin/cat
0804f000-08050000 rw-p 00006000 08:01 2514948    /bin/cat
08050000-08071000 rw-p 08050000 00:00 0          [heap]
b7d3b000-b7e75000 r--p 00000000 08:01 1475932    /usr/lib/locale/locale-archive
b7e75000-b7e76000 rw-p b7e75000 00:00 0
b7e76000-b7fcb000 r-xp 00000000 08:01 205950     /lib/i686/cmov/libc-2.7.so
b7fcb000-b7fcc000 r--p 00155000 08:01 205950     /lib/i686/cmov/libc-2.7.so
b7fcc000-b7fce000 rw-p 00156000 08:01 205950     /lib/i686/cmov/libc-2.7.so
b7fce000-b7fd1000 rw-p b7fce000 00:00 0
b7fe1000-b7fe3000 rw-p b7fe1000 00:00 0
b7fe3000-b7fe4000 r-xp b7fe3000 00:00 0          [vdso]
b7fe4000-b7ffe000 r-xp 00000000 08:01 196610     /lib/ld-2.7.so
b7ffe000-b8000000 rw-p 0001a000 08:01 196610     /lib/ld-2.7.so
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
```

- ~ 27 bits between bffeb000, b7ffee00.
- Top 4 not touched by PAX.
- < ~24 bits of randomness.
- Shacham et al report 16 bits in reality for x86 on Linux.

When to Randomize?

1. When the machine starts? (Windows)
   – Assign each module an address once per boot

2. When a process starts? (Linux)
   – Constant re-randomization for all child processes

Randomize!

Security Game for ASLR

- Attempted attack with randomization guess ***x*** is "a probe"
  - Success = **x** is correct
  - Failure = detectable crash or fail to exploit
  - Assume 16 bits of randomness available for ASLR

- **Game:**
  In expectation, how many probes are necessary to guess **x**?

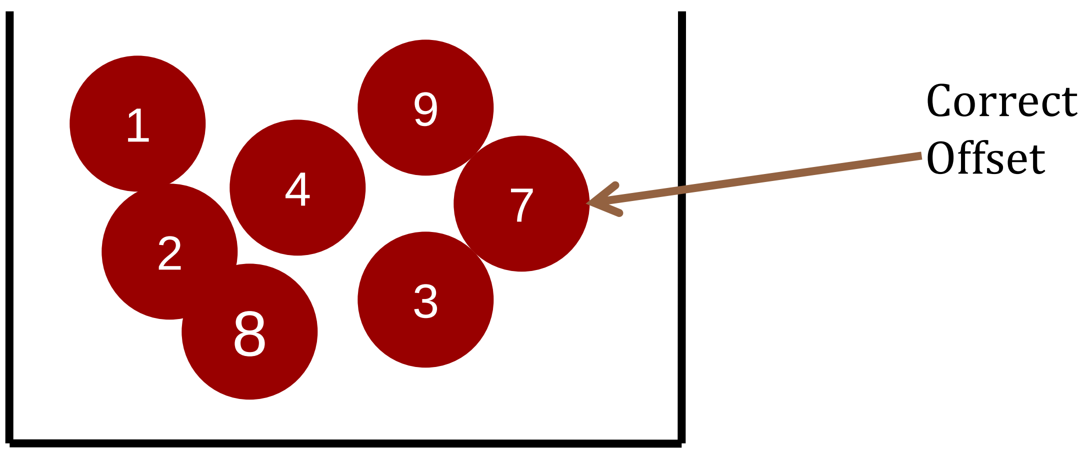
- *Scenario 1:* not randomized after each probe (Windows)
- *Scenario 2:* re-randomized after each probe (Linux)

What is the expected number of probes to hack the machine?

1. Pr[Success on exactly trial n]?
2. Pr[Success by trial n]?

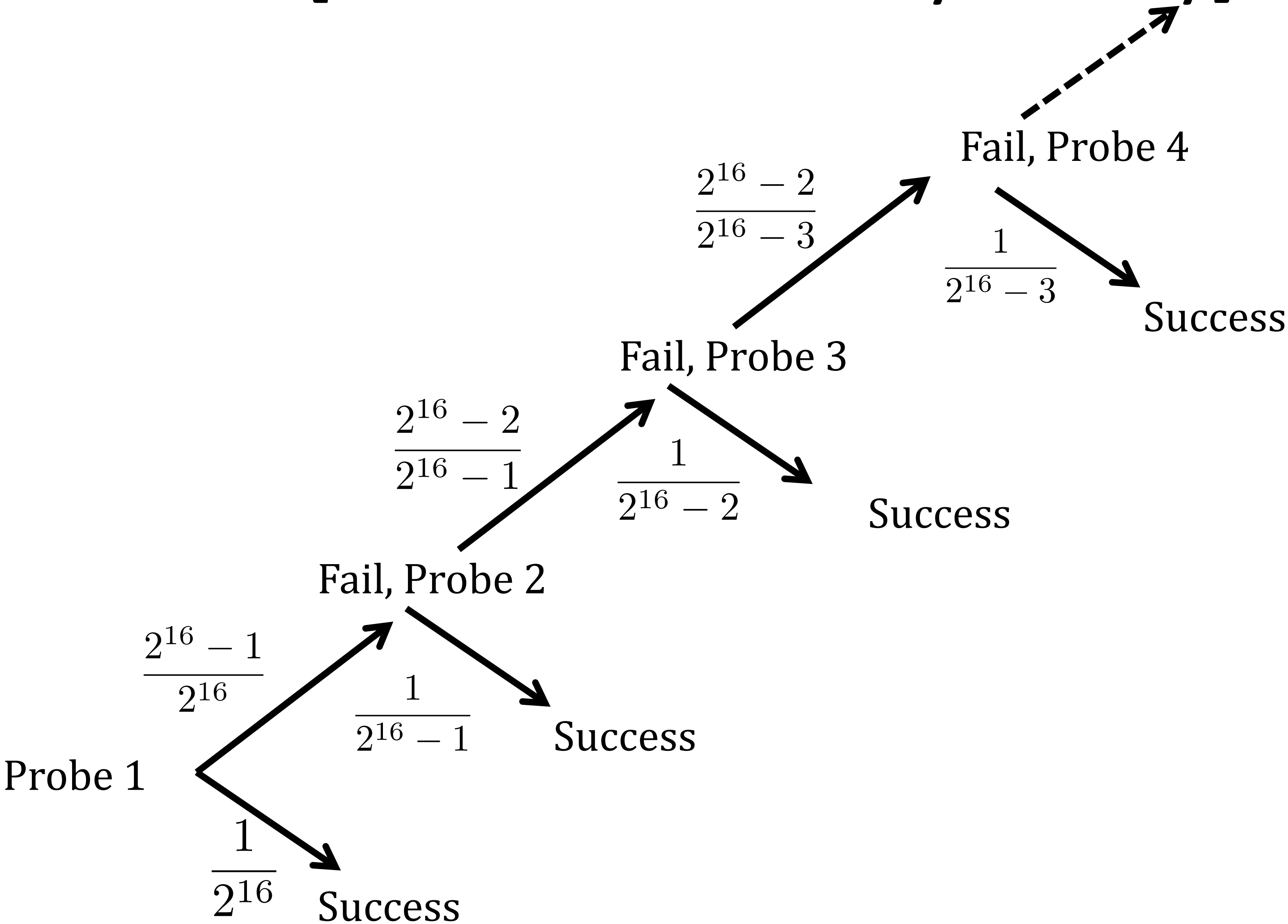
Scenario 1:
Not Randomized After Each Probe

- Pretend that each possible offset is written on a ball.
- There are 2^{16} balls.
- This scenario is like selecting balls ***without replacement*** until we get the ball with the randomization offset written on it.



W/O Replacement:

Pr[Success on Exactly nth try]



W/O Replacement:

Pr[Success on Exactly nth try]

$$\frac{2^{16}-1}{2^{16}} * \frac{2^{16}-2}{2^{16}-1} * \ldots * \frac{2^{16}-n-1}{2^{16}-n} * \frac{1}{2^{16}-n-1} = \frac{1}{2^{16}}$$

Fail the first n-1 times

Succeed on nth trial

W/O Replacement:

Pr[Success *by* nth try] =

Pr[Success on 1st try] +

Pr[Success on 2nd try] +

Pr[Success on nth try] = $\frac{n}{2^{16}}$

Expected Value

- E[X] is the expected value of random variable X
  - Basically a weighted average

$$\mathrm{E}[X] = x_1 p_1 + x_2 p_2 + \ldots + x_k p_k \, .$$

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} x_i \, p_i ,$$

Expected number of trials before success

Pr[success by nth try]

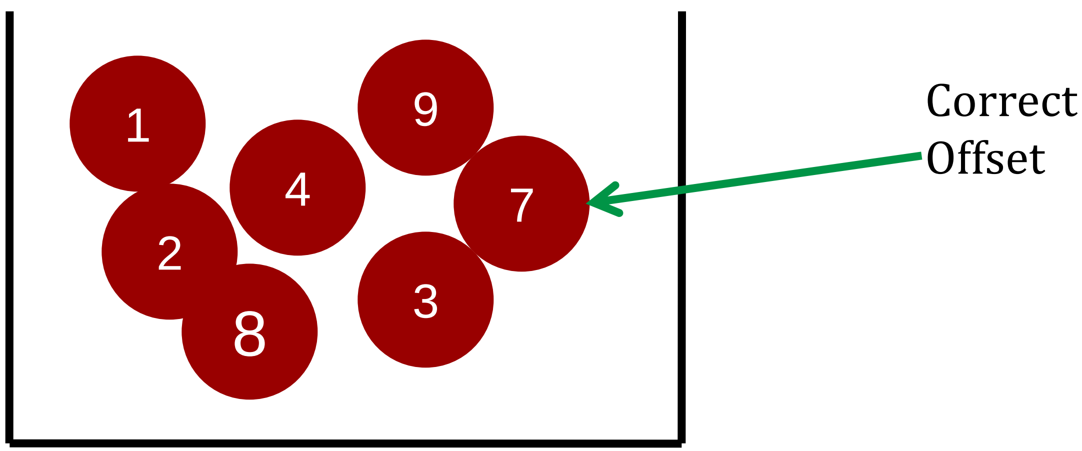
$$\text{Expectation} : \sum_{n=1}^{2^{16}} n * \underbrace{\frac{1}{2^{16}}}$$

$$= \frac{1}{2^{16}} * \sum_{n=1}^{2^{16}} n$$

$$= \frac{2^{16}+1}{2}$$

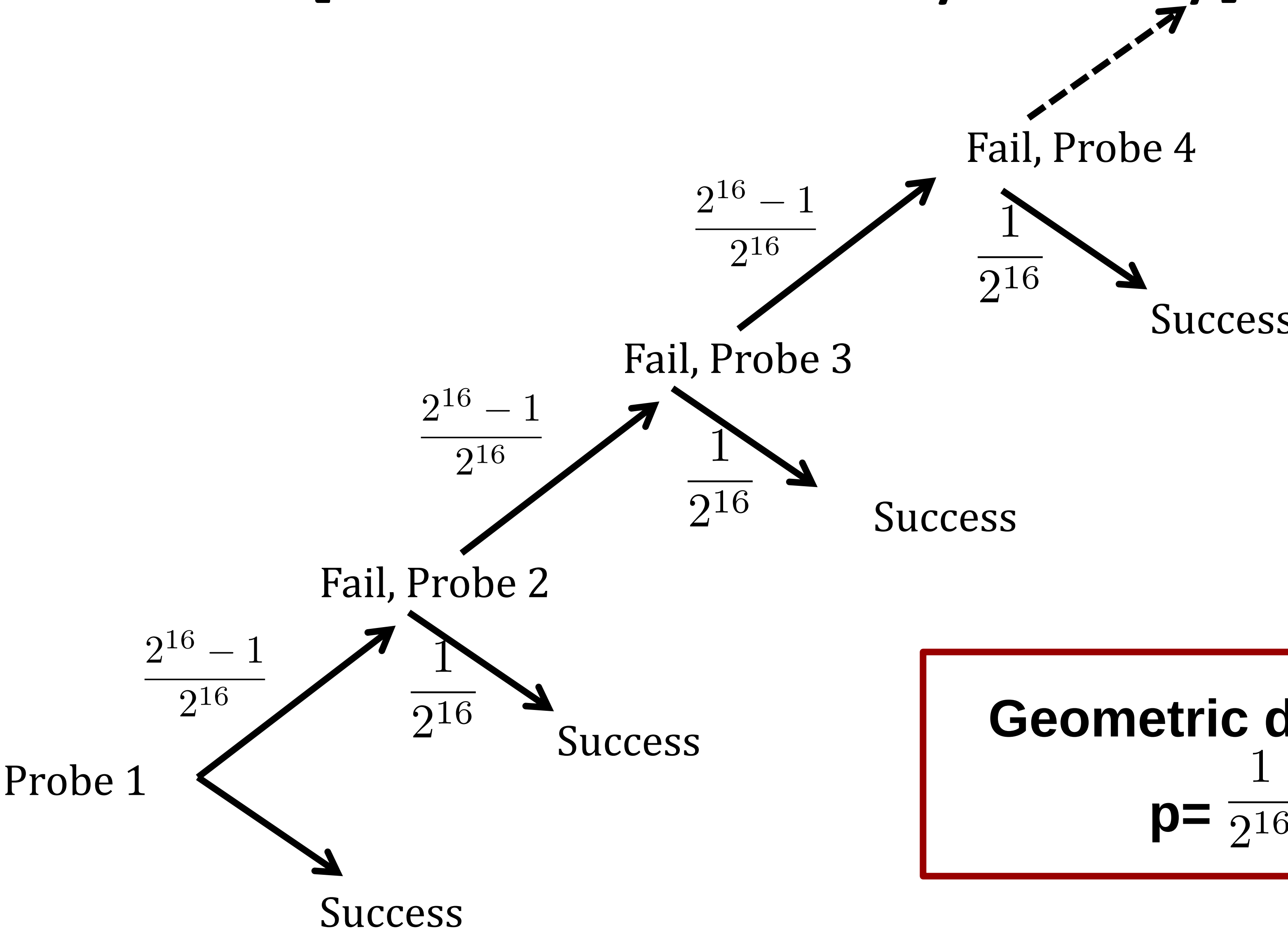
Scenario 2:
Randomized After Each Probe

- Pretend that each possible offset is written on a ball.
- There are 2^{16} balls.
- Re-randomizing is like selecting balls ***with replacement*** until we get the ball with the randomization offset written on it.



With Replacement

Pr[Success on exactly nth try]



Geometric dist.

p= $\frac{1}{2^{16}}$

With **Replacement:**

Expected number of probes: $1/p = 2^{16}$

$E[X] = 1/p$ for geometric distribution

$p = \frac{1}{2^{16}}$

Comparison

Expected success in 2^{16} probes

With Re-Randomization

For n bits of randomness: 2^n

Expected success in 2^{15} probes

Without Re-Randomization

For n bits of randomness: 2^{n-1}

Re-Randomization gives (only)1 bit of extra security!

But wait…

That's true, but is brute force the *only* attack?


Questions?

END

Backup slides here.

- Titled cherries because they are for the pickin. (credit due to maverick for wit)

Last Two Lectures

Control flow hijacks
are due to BUGS!

Format String Attacks

Microsoft took a drastic measure:

%n is disabled by default

- since VS 2005
- http://msdn.microsoft.com/en-us/library/ms175782(v=vs.80).aspx

- int _set_printf_count_output(
-    int enable
-);